UNITED STATES PATENT APPLICATION

for

COMPRESSION-DECOMPRESSION MECHANISM

Inventors:

Ali-Reza Adl-Tabatabai
Anwar M. Ghuloum

Prepared by:

BLAKELY, SOKOLOFF, TAYLOR & ZAFMAN LLP
12400 Wilshire Boulevard
Los Angeles, CA 90025-1026
(303) 740-1980

File No.: 042390.P17035

COMPRESSION-DECOMPRESSION MECHANISM

FIELD OF THE INVENTION

[0001]   The present invention relates to computer systems; more

particularly, the present invention relates to compressing data within a computer

system.

BACKGROUND

[0002]   Currently, various mechanisms are employed to compress data in

computer systems.   Such methods include adaptive dictionary based

algorithms. Dictionary based algorithms feature scanning a data block to be

compressed in order to find frequently used values (or redundancies). The

redundancies are replaced in the data block with pointers to various locations

within a dictionary table, where the value is stored. The dictionary and the

compressed data block are subsequently transmitted. Once received the data

block is decompressed by reinserting the redundant values in place of the

pointers.

[0003]   Existing dictionary-based compression methods (such as X-Match,

Wilson-Kaplan and the LZ variants) serially decompress each symbol in a

compressed block. Thus, random access into the compressed block is precluded.

The additional latency due to serial access makes existing dictionary-based

compression methods undesirable for latency-sensitive applications that require

fast random access.

BRIEF DESCRIPTION OF THE DRAWINGS

[0004]　　　　The present invention will be understood more fully from the detailed description given below and from the accompanying drawings of various embodiments of the invention. The drawings, however, should not be taken to limit the invention to the specific embodiments, but are for explanation and understanding only.

[0005]　　　　**Figure 1** illustrates one embodiment of a computer system;

[0006]　　　　**Figure 2** illustrates one embodiment of a compressed data block format;

[0007]　　　　**Figure 3** is a block diagram illustrating one embodiment of a cache controller;

[0008]　　　　**Figure 4** illustrates one embodiment of a compression data path;

[0009]　　　　**Figure 5** illustrates one embodiment of compression logic;

[0010]　　　　**Figure 6** illustrates another embodiment of compression logic;

[0011]　　　　**Figure 7** illustrates another embodiment of compression logic;

[0012]　　　　**Figure 8** illustrates one embodiment of decompression logic; and

[0013]　　　　**Figure 9** illustrates one embodiment of logic for a decompression unit.

DETAILED DESCRIPTION

[0014] A compression-decompression mechanism is described. In the following description, numerous details are set forth. It will be apparent, however, to one skilled in the art, that the present invention may be practiced without these specific details. In other instances, well-known structures and devices are shown in block diagram form, rather than in detail, in order to avoid obscuring the present invention.

[0015] Reference in the specification to "one embodiment" or "an embodiment" means that a particular feature, structure, or characteristic described in connection with the embodiment is included in at least one embodiment of the invention. The appearances of the phrase "in one embodiment" in various places in the specification are not necessarily all referring to the same embodiment.

[0016] **Figure 1** is a block diagram of one embodiment of a computer system 100. Computer system 100 includes a central processing unit (CPU) 102 coupled to bus 105. In one embodiment, CPU 102 is a processor in the Pentium® family of processors including the Pentium® II processor family, Pentium® III processors, and Pentium® IV processors available from Intel Corporation of Santa Clara, California. Alternatively, other CPUs may be used.

[0017] A chipset 107 is also coupled to bus 105. Chipset 107 includes a memory control hub (MCH) 110. MCH 110 may include a memory controller 112 that is coupled to a main system memory 115. Main system memory 115

stores data and sequences of instructions and code represented by data signals that may be executed by CPU 102 or any other device included in system 100.

[0018]   In one embodiment, main system memory 115 includes dynamic random access memory (DRAM); however, main system memory 115 may be implemented using other memory types. Additional devices may also be coupled to bus 105, such as multiple CPUs and/or multiple system memories.

[0019]   In one embodiment, MCH 110 is coupled to an input/output control hub (ICH) 140 via a hub interface. ICH 140 provides an interface to input/output (I/O) devices within computer system 100. For instance, ICH 140 may be coupled to a Peripheral Component Interconnect bus adhering to a Specification Revision 2.1 bus developed by the PCI Special Interest Group of Portland, Oregon.

[0020]   According to one embodiment, a cache memory 103 resides within processor 102 and stores data signals that are also stored in memory 115. Cache 103 speeds up memory accesses by processor 102 by taking advantage of its locality of access. In another embodiment, cache 103 resides external to processor 103.

[0021]   According to a further embodiment, cache 103 includes compressed cache lines to enable the storage of additional data within the same amount of area. In such an embodiment, the cache lines are compressed via a Parallel Dictionary Decompression (PDD) compression mechanism.

[0022]   In one embodiment, PDD is effective on program heap data and on

small block sizes (e.g., 64-128 bytes) by taking advantage of redundancies typically found in program data (e.g., redundancies in the upper bits of pointers and small integer values). PDD compresses a fixed-size block of data serially (e.g., one 4-byte dword or 8-byte chunk per clock).

[0023]   The result of compressing a block is a fixed-size compressed block with a size that depends on the compression ratio. In one embodiment, a compressed block includes a fixed number of compressed symbols (each of which is a compressed representation of a 32-bit word in the uncompressed block) and a fixed number of dictionary elements.

[0024]   **Figure 2** illustrates one embodiment of a PDD compressed data block format. The compressed block includes two dictionary elements (D0 and D1) and 16 compressed symbols (unmatched bits C0-C15 and tags T0-T15). To enable parallel decompression, PDD compresses blocks such that dictionary elements and compressed symbols are a fixed length and at a fixed offset within the compressed block.

[0025]   Tags within a compressed symbol indicate a type of decompression being used. **Table 1** shows an example encoding for the tags in the compressed block illustrated in **Figure 2**. A 2-bit tag T_i encodes 4 possible ways in which the corresponding ith symbol is decompressed.

[0026]   If $T_i = 00$, a 0-extension of the unmatched bits C_i occurs. For example, if T15 is 0 and C15 is 1, the first word is 1, which is preceded by all zeroes. If $T_i = 01$, a 1-extension of the unmatched bits C_i occurs. For example, if

T15 is 1 and C15 is 1, the first word has a negative value (depending on the width of C), which is preceded by all ones. If $Ti = 10$, the unmatched bits Ci are appended to the bits of dictionary element D0. Similarly, if $Ti = 11$, the unmatched bits Ci are appended to the dictionary element D1.

Table 1

Ti	Decompression method
00	0 extend unmatched bits
01	1 extend unmatched bits
10	Append unmatched bits to D0
11	Append unmatched bits to D1

[0027]   In contrast to existing compression mechanisms, which have a variable compression ratio to compress by as much as possible, PDD has a fixed compression ratio. A fixed compression ratio suits applications that manage memory in fixed chunks and require fast decompression latency. For instance, cache memory is organized and managed in 64 or 128-byte sectors so that variable decompression ratio leads to fragmentation (e.g., unused space in the compressed block). Although described with reference to a cache compression application, one of ordinary skill in the art will appreciate that the PDD compression mechanism may be implemented in other applications (e.g., such as memory and bus compression, and network packet compression).

[0028]   The compression ratio of PDD depends on several design parameters including the size of the block being compressed, the number of dictionary elements, and the size of each dictionary element. The design

parameters can be tuned to meet the compression ratio requirements of the target application for which compression is being used, and to maximize the number of blocks compressed in the target workloads.

[0029]   **Figure 3** illustrates one embodiment of cache controller 104. Cache controller 104 includes compression logic 310 and decompression logic 320. Compression logic 310 implements the PDD mechanism to compress data blocks. **Figure 4** illustrates one embodiment of a compression data path. The compression data path includes registers (RS), logic 420 and buffer 430.

[0030]   According to one embodiment, PDD compresses one 32-bit symbol per clock cycle. At iteration i, the ith symbol S^i (held in register RS) is split into its upper 21 bits (signal U^i) and its bottom 11 bits (the unmatched bits C^i). U^i is compressed into a tag T^i, which is accumulated along with C^i in a buffer. Registers RD0 and RD1 hold the two dictionary elements and registers RV0 and RV1 are Booleans that indicate whether RD0 and RD1 hold valid dictionary elements, respectively.

[0031]   At iteration i, signal D_j^i is the value of dictionary element RDj and is valid only if signal V_j^i is true. The initial value of RVj is false, and the initial value of RDj is zero. At each iteration i, logic 420 takes as input the dictionary values D_j^i, dictionary valid bits V_j^i, and upper bits of the symbol U^i, and produces the tag T^i for the current iteration as well as the dictionary values D_j^{i+1} and valid bits V_j^{i+1} for the next iteration (i.e., iteration i+1).

[0032]   In one embodiment, the RV and RD registers load new values upon

each iteration. The not compressible signal (NC) is set to true, if U^i is not compressible (e.g., U^i cannot be compressed via sign extension, it does not match any values in the dictionary elements, and the dictionary elements are all valid).

[0033] After 16 iterations, the buffer holds the 16 compressed symbols (208 bits of data), and the dictionary registers, RD0 and RD1, hold the dictionary elements. The dictionary registers and buffer 430 are combined to form the compressed block, regardless of the values in RV0 and RV1 (sometimes dictionary elements are unused in a compressed block, indicated by a false value in RV0 or RV1).

[0034] **Figure 5** illustrates one embodiment of logic 420. Logic 420 includes dictionary comparison logic 505, match logic 510, no match logic 520 and tag encoder 550. Match logic 510 determines if there is a match, resulting in successful compression for a particular iteration. For instance the upper 21 bits of word are compared against each dictionary at dictionary comparison logic 505. If there is a match, tag encoder compresses the data, as will be described below.

[0035] The and-gate and nor-gate in logic 510 determine whether the bits are all ones, or all zeroes, respectively. If all ones, the data is compressed via one extension. If all zeroes, the data is compressed via zero extension. If the bits are not all ones, all zeroes, or do not match any of the dictionary elements, a no match signal is transmitted to no match logic 520. No match logic 520 is used to store the unmatched bits in the next dictionary entry. One of ordinary skill in the

art will appreciate that other types of logic circuitry may be used to implement the components of logic 420.

[0036]   Tag encoder 550 uses the match, sign-extension, and valid signals to generate the tag value according to the encoding of **Table 1**. **Table 2** shows a truth table for tag encoder 550.

Table 2

S_F	S_T	M_0	M_1	V_0	T_1	T_0	
1	-	-	-	-	0	0	0-extend
-	1	-	-	-	0	1	1-extend
0	0	1	-	-	1	0	D0
0	0	0	1	-	1	1	D1
0	0	0	0	0	1	0	D0
0	0	0	0	1	1	1	D1

[0037]   In one embodiment, the critical path in **Figure 5** can be reduced by performing tag encoding in a separate pipeline stage (removing it altogether from the critical path), and by overlapping generation of the previous iteration's valid bits with the matching logic (which makes the critical path be the maximum of either the match logic delay or the generation of the valid bits).

[0038]   **Figure 5** illustrates compressing one 32-bit symbol per clock cycle. However in other embodiments, more than one, for example, two 32-bit symbols (a "chunk") may be compressed at a time, allowing data that arrives over an 8-byte bus to be compressed as it arrives. **Figure 6** illustrates another embodiment of logic 420 for compressing a chunk at a time.

[0039]   In one embodiment, the number of dictionary elements may be varied. **Figure 7** illustrates one embodiment of logic 420 implementing k

dictionary elements. In one embodiment, the number of dictionary elements (N_D) is quantitatively related to several parameters such as a number of leading bits matched (L), block size (B) in bits, size of compression tags (T) and word size (W). In a further embodiment, the number of leading bits can be calculated based upon the following equations:

$$L * N_D + \frac{B}{W} * (T + (W - L) + \lceil \log_2 N_D \rceil) \leq \frac{B}{2} \text{ if } N_D > 1; \text{ and}$$

$$L * N_D + \frac{B}{W} * (T + (W - L)) \leq \frac{B}{2} \text{ if } N_D \leq 1$$

[0040] Therefore, using PDD enables picking a fixed number of leading bits to match and automatically derive the number of dictionary elements available. In another embodiment, the number of desired dictionary elements can be fixed in order to solve for the leading bits allowed in partial matches and sign extension.

[0041] According to other embodiments, the format of a compressed block can also be varied. For example, the dictionary elements can be placed in the middle of the compressed block or at either ends of the compressed block. If the compressed block is transmitted serially over a bus, then placing the dictionary elements at the beginning of the compressed block allows decompression to be overlapped with arrival of the compressed data.

[0042] If the compressed block is available in parallel, then placing the dictionary elements in the middle of the block minimizes delays in distributing

the elements to the decompression units. In a further embodiment, the

dictionary elements may be replicated throughout the compressed block.

Replicating the dictionary elements to provide efficient access to all segments of

the block.

[0043]   In another embodiment, different methods of combining

unmatched bits with dictionary elements may be implemented, as well as

different methods of sign-extending unmatched bits to handle data types such as

packed 8 or 16-bit integers, unicode characters (Utf16), aligned pointers, and

floating point. For example, the compression logic can divide a 32-bit dword

into 2 16-bit halves and compress each half's leading sign bits. Compression can

also be combined with power optimizations by inverting the dictionary elements

and unmatched bits to maximize zeroes. The inversion can be encoded in the

tags.

[0044]   Referring back to **Figure 3**, decompression logic 320 decompresses

a data block once the block is received at its destination. In one embodiment,

decompressor 320 implements PDD to decompress symbols in a compressed

block in parallel. To decompress a symbol, PDD either sign-extends its

unmatched bits or combines its unmatched bits with the bits in one of the

dictionary elements. A symbol's tag indicates whether the symbol's unmatched

bits should be sign-extended or combined with a dictionary element. If the

symbol is to be combined with a dictionary element, the tag indicates the index

of the dictionary element as well as how the unmatched bits and dictionary

element are combined.

[0045]   **Figure 8** illustrates one embodiment of decompression logic 320.

Decompression logic 320 includes a decompression units 820 associated with

each compressed symbol. The decompression units 820 operate in parallel. Each

decompression unit 820 takes as input a compressed symbol (Ti and Ci), and the

two dictionary elements D0 and D1, and produces as output a 32-bit

decompressed symbol Si.

[0046]   The latency to produce a decompressed symbol Si equals the delay

to distribute the dictionary elements D0 and D1 to Si's decompression unit, plus

the latency of the decompression unit. In one embodiment, unmatched bits are

each 11 bits; therefore, dictionary elements are each 21 bits, and the compressed

block is 250 bits. The decompressed block is 512 bits for a compression ratio of

slightly better than 2:1. Thus, such an embodiment is suitable for compressing 64

byte data, such as cache lines, down to 32 bytes. However, one of ordinary skill

in the art will appreciate that other size data blocks, dictionary elements and

compression ratios may be implemented without departing from the true scope

of the invention.

[0047]   **Figure 9** illustrates one embodiment of logic for a decompression

unit 820. The unmatched bits are passed through to form the least significant 11

bits of the uncompressed symbol. Decompression unit 820 implements 2 levels

of 2-input multiplexers wherein the tag bits select the most significant 21 bits of

the uncompressed symbol according to the encoding shown above in **Table 1**.

[0048]   The PDD mechanism enables dictionary based data blocks to be decompressed in parallel, thus various data within the block may be randomly decompressed and access without having to wait for the entire block to be decompressed. Accordingly, latency-sensitive applications, such as cache line compression, may implement PDD without incurring performance losses.

[0049]   Whereas many alterations and modifications of the present invention will no doubt become apparent to a person of ordinary skill in the art after having read the foregoing description, it is to be understood that any particular embodiment shown and described by way of illustration is in no way intended to be considered limiting. Therefore, references to details of various embodiments are not intended to limit the scope of the claims which in themselves recite only those features regarded as the invention.